Chapter 5

The $\mathbb{K}$ Semantic Framework

This chapter introduces $\mathbb{K}$, a rewriting-based executable semantic framework which will be used in the remainder of this book. $\mathbb{K}$ was first introduced by the author in the lecture notes of a programming language design course at the University of Illinois at Urbana-Champaign (UIUC) in Fall 2003 [64], as a means to define concurrent languages in rewriting logic using Maude. Since 2003, $\mathbb{K}$ has been used continuously in teaching programming languages, in seminars, and in research.

Programming languages, calculi, as well as type systems or formal analysis tools can be defined in $\mathbb{K}$ by making use of *configurations*, *computations* and *rules*. Configurations organize the system/program state in units called cells, which are labeled and can be nested. Computations are special structures which carry "computational meaning". More precisely, computations are nested list terms which sequentialize computational tasks, such as fragments of program; in particular, computations extend the original programming language or calculus syntax. $\mathbb{K}$ (rewrite) rules generalize conventional rewrite rules by making it explicit which parts of the term they read-only, write-only, or do not care about. This distinction makes $\mathbb{K}$ a suitable framework for defining truly concurrent languages or calculi even in the presence of sharing. Since computations can be handled like any other terms in a rewriting environment, that is, they can be matched, moved from one place to another in the original term, modified, or even deleted, $\mathbb{K}$ is particularly suitable for defining control-intensive language features such as abrupt termination, exceptions or call/cc.

The $\mathbb{K}$ framework consists of two components:

1. The $\mathbb{K}$ *concurrent rewrite abstract machine*, abbreviated KRAM and discussed in Section 5.4;

2. The $\mathbb{K}$ *technique*, discussed in Section 5.5.

Like conventional rewrite systems, a $\mathbb{K}$-system consists of a signature for building terms and of a set of rules for iteratively rewriting terms. Like in rewriting logic (Section 2.7), $\mathbb{K}$ rules can be applied concurrently and unrestricted by context. The novelty of the KRAM (Section 5.4) is that its rules contain, besides expected information saying how the original term is modified (the *write data*), also information about what parts of the term are shared with other rules (the *read-only data*). This additional information provided as part of the rules allows $\mathbb{K}$ to be a suitable rewrite-based framework for semantically defining truly concurrent programming languages and calculi whose threads or processes may be desired to share data. The $\mathbb{K}$ concurrent rewrites associated to a $\mathbb{K}$ system may require several interleaved rewrites in the rewriting logic theory straightforwardly associated to the $\mathbb{K}$-system (i.e., by forgetting the sharing information).

Even though the KRAM aims at maximizing the amount of concurrency that can be achieved in a rewriting setting, it does not tell *how* one can define a programming language or a calculus as a $\mathbb{K}$-system. In particular, a bad $\mathbb{K}$ definition may partially or totally inhibit KRAM's potential for concurrency. The $\mathbb{K}$ technique discussed in Section 5.5 proposes a definitional methodology that makes the use of the KRAM convenient when formally defining programming languages and calculi. Moreover, the $\mathbb{K}$ technique can also be and actually has already been intensively used as a technique to define languages and calculi as conventional term rewrite systems or as rewriting logic theories. There is one important thing lost in the translation of $\mathbb{K}$ into rewriting logic though, namely the degree of true concurrency of the original $\mathbb{K}$-system. Ignoring this true concurrency aspect, the relationship between $\mathbb{K}$ and rewriting logic in general and Maude in particular is the same as that between any of the conventional semantic styles discussed in Chapter 3 and rewriting logic and Maude: the latter can be used to execute and analyze $\mathbb{K}$-systems.

This chapter is structured as follows:

- Section 5.1 discusses informally the requirements that have led to the design and development of the $\mathbb{K}$ framework, and hereby, it highlights the objectives that $\mathbb{K}$ attempts to achieve. These requirements are derived from what we believe the characteristics of an ideal semantic framework should be, and the motivation for the $\mathbb{K}$ framework comes from the observation that the existing semantic frameworks fail to satisfy these requirements.

- Section 5.2 gives a quick overview of the $\mathbb{K}$ framework by using it to define the IMP language (Section 3.1) and its extension IMP++ (Section 3.8). This section should give the reader quite a clear feel for what $\mathbb{K}$ is about and how it operates.

- Section 5.3 shows how $\mathbb{K}$ definitions can be represented in rewriting logic and then efficiently executed using Maude. The role played by this section is the same as that played by the similar sections corresponding to various programming language definitional styles in Chapter 3.

- Section 5.4 describes the $\mathbb{K}$ concurrent rewrite abstract machine (KRAM) and is the most technical part of this chapter. However, it formalizes a relatively intuitive process of concurrent rewriting with sharing, so the reader interested more in using $\mathbb{K}$ then in the technical details of its core concurrent rewriting machinery may safely skip this section.

- Section 5.5 presents the $\mathbb{K}$ technique in detail, explaining essentially how the KRAM or other rewrite infrastructures can be used to define programming language semantics by means of nested-cell configurations, computations and rewrite rules.

- Section 5.7 shows $\mathbb{K}$ at work: it introduces and at the same time shows how to give a $\mathbb{K}$ semantics to CHALLENGE, a programming language containing varied features known to be problematic to define in other frameworks. CHALLENGE was conceived as a means to challenge the various language definitional frameworks and to expose their limitations.

5.1 Quest for an Ideal Language Definitional Framework

Chapter 3 showed that any conventional language definitional style can be faithfully, step-for-step, captured by a rewriting logic theory. It may then seem "obvious" to the hasty reader that rewriting logic is perhaps the ideal language definitional framework and thus naturally ask the following:

What is the need for yet another language definitional framework that can be embedded in rewriting logic, $\mathbb{K}$ in this case, if rewriting logic is already so powerful?

Unfortunately, in spite of its definitional strength as a computational logic framework, rewriting logic does not give, and does not intend to give, the programming language designer any recipe on *how* to define a language. It essentially only suggests the following: however one wants to formally define a programming language or calculus, one can probably also do it in rewriting logic following the same intuitions and style. Therefore, rewriting logic can be regarded as a *meta-framework* that supports definitions of programming languages and calculi among many other things, providing the language designer with a means to execute and formally analyze languages in a generic way, but only *after* the language is already defined. Additionally, as discussed in Section 2.7 and in more depth in Section 5.2, the natural rewriting logic definition of a concurrent programming language may enforce interleaving in situations where true concurrency is meant.

The introduction and development of $\mathbb{K}$ was largely motivated by the observation that after more than 40 years of systematic research in programming language semantics, the following important (multi-)question remains largely open to the working programming language designer, and not only:

Is there any language definitional framework that, at the same time,

1. *Gives a strong intuition, even precise recipes, on how to define a language?*
2. *Same for language-related definitions, such as type checkers, type inferencers, abstract interpreters, safety policy or domain-specific checkers, etc.?*
3. *Can define arbitrarily complex language features, including, obviously, all those found in existing languages, capturing also their intended computational granularity?*
4. *Is modular, that is, adding new language features does not require to modify existing definitions of unrelated features? Modularity is crucial for scalability and reuse.*
5. *Supports non-determinism and concurrency, at any desired granularity?*
6. *Is generic, that is, not tied to any particular programming language or paradigm?*
7. *Is executable, so one can "test" language or formal analyzer definitions, as if one already had an interpreter or a compiler for one's language? Efficient executability of language definitions may even eliminate the need for interpreters or compilers.*
8. *Has state-exploration capabilities, including exhaustive behavior analysis (e.g., finite-state model-checking), when one's language is non-deterministic or/and concurrent?*
9. *Has a corresponding initial-model (to allow inductive proofs) or axiomatic semantics (to allow Hoare-style proofs), so that one can formally reason about programs?*

The list above contains a *minimal* set of desirable features that an ideal language definitional framework should have. Unfortunately, the current practice is to take the above features one at a time, temporarily or permanently declaring the others as "something else". We next describe how current practices and language definitional styles fail to satisfy the above-mentioned requirements.

1. *Gives a strong intuition, even precise recipes, on how to define a language?*

   To formalize one's intuition about a language feature, it is common practice to use a big-step or a small-step SOS definition, with or without evaluation contexts, typically on paper, without any machine support. Sometimes this so-called "formal" process is pushed to extreme in what

regards its informality, in the sense that one can see definitions of some language features using one definitional style and of other features using another definitional style, without ever proving that the two definitional styles can co-exist in the claimed form for the particular language under consideration. For example, one may use a big-step SOS to give semantics to a code-self-generation extension of Java, while using a small-step SOS to define the concurrency semantics of Java. However, common sense tells that once one has concurrency and shared memory, one cannot have a big-step SOS definition. An ideal language definitional framework should provide a uniform, compact and rigorous way to modularly define various language features, avoiding the need to define different language features following different styles.

2. *Same for language-related definitions, such as type checkers, type inferences, abstract interpreters, safety policy or domain-specific checkers, etc.?*

   To define a type system or a (domain-specific or not) safety policy for a language, one may follow a big-step-like definitional style, or even simply provide an algorithm to serve as a formal definition. While this appears to be, and in many cases indeed is acceptable, there can be a significant "formal gap" between the actual language semantic definition and its type system or safety policy regarded as mathematical objects, because in order to carry out proofs relating the two one needs one common formal ground. In practice, one typically ends up "encoding" the two in yet another framework, claimed to be "richer", and then carry out the proofs within that framework. But how can one make sure that the encodings are correct? Do they serve as alternative definitions for that sole purpose?

   An ideal language definitional framework should have all the benefits of the "richer" framework, at no additional notational or logical complexity, yet naturally capturing the complete meaning of the defined constructs. In other words, in an ideal framework one should define a language as a mathematical object, say $\mathcal{L}$, and a type system or other abstract interpretation of it as another mathematical object over the same formalism, say $\mathcal{L}'$, and then carry out proofs relating $\mathcal{L}$ and $\mathcal{L}'$ using the provided proof system of the definitional framework. $\mathcal{L}$, $\mathcal{L}'$, as well as other related definitions, should be human readable and easy to understand enough so that one does not feel the drive to give alternative, more intuitive definitions using a more informal notation or framework.

3. *Can define arbitrarily complex language features, including, obviously, all those found in existing languages, capturing also their intended computational granularity?*

   Some popular language definitional frameworks are incapable of defining even existing language features. The fact that a particular language feature is supported in some existing language serves as the strongest argument that that feature may be desirable, so an ideal language definitional framework must simply support it; in other words, one cannot argue against the usefulness of that feature just because one's favorite definitional framework does not support it. For example, since in standard SOS definitions (not including reduction semantics with evaluation contexts) the "control flow" information of a program is captured within the structure of the "proof", and since proof derivations are not first class objects in these formalisms, it makes it very hard, virtually impossible in these formalisms to define complex control intensive language constructs like, e.g., call-with-current-continuation (callcc).

   Another important example showing that conventional definitional frameworks (e.g., SOS) fail to properly support existing common language features, is concurrency. Most frameworks

enforce an interleaving semantics, which may not necessarily always be the desired approach to concurrency. In particular, an implementation of a multi-threaded system in which two threads can concurrently read a shared variable would be dissallowed, because it disobeys the "formal" interleaving-based language definition. Concurrency is further discussed in item 5.

Some frameworks provide a "well-chosen" set of constructs, shown to be theoretically sufficient to define any computable function or algorithm, and then propose *encodings* of other language features into the set of basic ones; examples in this category are Turing machines or the plethora of (typed or untyped) λ-calculi, or π-calculi, etc. While these basic constructs yield interesting and meaningful idealized programming languages, using them to encode other language features is, in our view, inappropriate. Indeed, encodings hide the intended *computational granularity* of the defined language constructs; for example, a variable lookup intended to be a one-step operation in one's language should take precisely one step in an ideal framework (not hundreds/thousands of steps as in a Turing machine or lambda calculus encoding, not even two steps: first get location, then get value).

4. *Is modular, that is, adding new language features does not require to modify existing definitions of unrelated features? Modularity is crucial for scalability and reuse.*

   As Mosses pointed out in [57] and as shown in Chapter 3 and discussed in Section 3.9, big-step and small-step SOS are non-modular; Plotkin himself had to modify the definition of simple arithmetic expressions (in the original notes on SOS [63]) three times as his initial language evolved. As seen in Section 3.8, to add an innocent abrupt termination statement to a language defined using SOS, say a `halt`, one needs to more than double the total number of rules: each language construct needs to be allowed to "propagate" the halting signal potentially generated by its arguments. Also, as one needs to add more items into configurations to support new language features, in SOS one needs to change again every rule to include the new items; note that there are no less than $4 + m * 4 + a * (8 + t * 4)$ configuration items in the configuration of CHALLENGE in Section 5.7 (which is a comparatively toy language), where m is the number of pending messages, a is the number of current agents, and t is the average number of threads within each agent. It can easily become very annoying and error prone to modify a large portion of unrelated existing definitions when adding a new feature.

   A language designer may be unwilling to add a new feature or improve the definition of an existing one, just because of the large number of required changes. Informal writing conventions are sometimes adopted to circumvent the non-modularity of SOS. For example, as already mentioned in Section 3.2.5, Milner and his collaborators propose a "store convention" in the definition of Standard ML [54] to avoid having to mention the store in every rule, and an "exception convention" to avoid having to double the number of rules for the sole purpose of supporting exceptions. As rightfully noticed by Mosses [57], such conventions are not only adhoc and language specific, but may also lead to erroneous definitions. Mosses' Modular SOS [57] (MSOS) brings modularity to SOS in a formal and elegant way, by grouping the non-syntactic configuration items into transition labels, and allowing rules to mention only those items of interest from each label. As discussed in Section 3.9, MSOS still inherits all the remaining limitations of SOS.

5. *Supports non-determinism and concurrency, at any desired granularity?*

   By inherently enforcing an interleaving semantics for concurrency, existing reduction semantics

definitions (including ones based on evaluation contexts) can only capture a projection of concurrency (when one's goal is to define a truly concurrent language), namely its resulting non-determinism. Proponents of existing reduction semantics approaches may argue that the resulting non-deterministic behavior of a concurrent system is all what matters, while proponents of true concurrency may argue that a framework which does not support naturally concurrent actions, i.e., actions that take place *at the same time*, is not a framework for concurrency. We do not intend to discuss the (admittedly important but debatable) distinctions between non-determinism and interleaving vs. true concurrency here. The fact that there are language designers who desire an interleaving semantics while others who desire a true concurrency semantics for their language is strong evidence that an ideal language definitional framework should simply support both, preferably with no additional settings of the framework, but rather via particular definitional methodologies within the framework.

6. *Is generic, that is, not tied to any particular programming language or paradigm?*

   A non-generic framework, i.e., one building upon a particular programming language or paradigm, may be hard or impossible to use at its full strength when defining a language that crosses the boundaries of the underlying language or paradigm. For example, a framework enforcing object or thread communication via explicit send and receive messages may require artificial encodings of languages that opt for a different communication approach (e.g., shared memory), while a framework enforcing static typing of programs in the defined language may be inconvenient for defining dynamically typed or untyped languages. In general, a framework providing and enforcing particular ways to define certain types of language features would lack genericity. Within an ideal framework, one can and should develop and adopt methodologies for defining certain types of languages or language features, but these should not be enforced. This genericity requirement is derived from the observation that today's programming languages are so diverse and based on orthogonal, sometimes even conflicting paradigms, that, regardless of how much we believe in the superiority of a particular language paradigm, be it object-oriented, functional or logical, a commitment to any existing paradigm would significantly diminish the strengths of a language definitional framework.

7. *Is executable, so one can "test" language or formal analyzer definitions, as if one already had an interpreter or a compiler for one's language? Efficient executability of language definitions may even eliminate the need for interpreters or compilers.*

   Most existing language definitional frameworks are, or until relatively recently were, lacking tool support for executability. Without the capability to execute language definitions, it is virtually impossible to debug or develop large and complex language definitions in a reasonable period of time. The common practice today is still to have a paper definition of a language using one's desired formalism, and *then* to implement an interpreter for the defined language following in principle the paper definition. This approach, besides the inconvenience of having to define the language twice, guarantees little to nothing about the appropriateness of the formal, paper definition. Compare this approach to an approach where there is *no gap* between the formal definition and its implementation as an interpreter. While any definition is by definition correct, one gets significantly more confidence in the appropriateness of a language definition, and is less reluctant to change it, when one is able to run it *as is* on tens or hundreds of programs. Recently, executability engines have been proposed both for MSOS (the MSOS tool, implemented by Braga and collaborators in Maude [18]) and for reduction semantics

with evaluation contexts (the PLT Redex tool, implemented by Findler and his collaborators in Scheme [37]). A framework providing *efficient* support for executability of formal language definitions may eliminate entirely the need to implement interpreters, or type checkers or type inferencers, for a language, because one can use directly the formal definition for that purpose.

8. *Has state-exploration capabilities, including exhaustive behavior analysis (e.g., finite-state model-checking), when one's language is non-deterministic or/and concurrent?*

   While executability of language definitions is indispensable when designing non-trivial languages, one needs richer tool support when the language is concurrent or non-deterministic. Indeed, it may be that one's definition is appropriate for particular thread or process interleavings (e.g., when blocks are executed atomically), but that it has unexpected behaviors for other interleavings. Moreover, somewhat related to the desired computational granularity of language constructs mentioned in item 3 above, one may wish to exhaustively investigate all possible interleavings or executions of a particular concurrent program, to make sure that no undesired behaviors are present and no desired behaviors are excluded. When the state space of the analyzed program is large, manual analysis of behaviors may not be feasible; therefore, model-checking and/or safety property analysis (through systematic state-space exploration) are also desirable as intrinsic components of an ideal language definitional framework.

9. *Has a corresponding initial-model (to allow inductive proofs) or axiomatic semantics (to allow Hoare-style proofs), so that one can formally reason about programs?*

   To prove properties about programs in a defined programming language, or properties about the programming language itself, as also mentioned in item 2 above, the current practice is to encode/redefine the language semantics in a "richer" framework, such as a theorem prover, and then carry out the desired proofs there. Redefining the semantics of a fully fledged programming language in a different formalism is a highly nontrivial, error prone and tedious task, possibly taking months; automated translations may be possible when the original definition of the language is itself formal, though one would need to validate the translator. In addition to the "formal gap" mentioned in item 2 due to the translation itself, this process of redefining the language is simply inconvenient. An ideal language definitional framework should allow one to have, for each language, "one definition serving all purposes", including all those mentioned above.

   Most successful current program verification approaches are based on axiomatic semantics (in the style of Hoare logic) of the language under consideration and on implementations of it in program verifiers. In fact, implementing program verifiers is still an art, one that few master. Existing program verifiers are based "in principle" on some implicit axiomatic semantics which is hand-crafted in the prover; a formal semantics is in fact not required and typically not given at all, thus creating an obvious gap between the implementation of a program verifier and the language semantics, that is, the language itself. Moreover, since axiomatic semantics are not executable and thus not testable, the underlying axiomatic semantics can be itself untrustable; at minimum, an alternative executable semantics of the language is needed and a proof that the axiomatic semantics is sound for it. Thus there is a double gap between a language definition and a program verifier for it. The very fact that one needs various semantics of a language for various purposes shows that none of these semantics is "ideal": as already stated above, an ideal language semantic definition should serve all the purposes.

There are additional desirable, yet of a more subjective nature and thus harder to quantify, requirements of an ideal language definitional framework. For example, it should be simple and easy to understand, teach and use by mainstream enthusiastic language designers, not only by language experts —in particular, an ideal framework should not require its users to have advanced concepts of category theory, logics, or type theory, in order to use it. Also, it should have good data representation capabilities and should allow proofs of theorems about programming languages that are easy to comprehend. Additionally, a framework providing support for parsing programs directly in the desired language syntax may be desirable to one requiring the implementation of an additional, external to the definitional setting, parser.

The nine requirements above are nevertheless ambitious. Some proponents of existing language definitional frameworks may argue that their favorite framework has these properties; however, a careful analysis of existing language definitional frameworks, like the one in Section 3.9, reveals that they actually fail to satisfy some of these ideal features. Others may argue that their favorite framework has some of the properties above, the "important ones", declaring the other properties either "not interesting" or "something else". For example, one may say that what is important in one's framework is to get a dynamic semantics of a language, but its (model-theoretical) algebraic denotational semantics, proving properties about programs, model checking, etc., are "something else" and therefore are allowed to need a different "encoding" of the language. Our position is that an ideal language definitional framework should not compromise any of the nine requirements above.

Whether $\mathbb{K}$ satisfies all the requirements above or not is, and probably will always be, open. What we can mention with regards to this aspect, though, is that $\mathbb{K}$ was motivated and stimulated by the observation that the existing language definitional frameworks fail to fully satisfy these minimal requirements; consequently, $\mathbb{K}$'s design and development were conducted aiming *explicitly* to fulfill all nine requirements discussed above, promoting none of them at the expense of others.

5.2 $\mathbb{K}$ Overview by Example

Here we briefly describe the $\mathbb{K}$ framework, what it offers and how it can be used. We use as concrete examples the IMP language (Section 3.1) and its extension IMP++ (Section 3.8), discussed in Chapter 3 in the context of the various existing language definitional frameworks. We define both an executable semantics and a type system for these languages. The type system is included mainly for demonstration purposes, to show that one can use the same definitional framework, $\mathbb{K}$, to define both formal language semantics and language abstractions. The role of this section is threefold: first, it gives the reader a feel for the $\mathbb{K}$ framework before we proceed to define it rigorously in the remainder of this chapter; second, it shows how $\mathbb{K}$ avoids the limitations of the various more conventional semantic approaches discussed in Chapter 3; and third, it shows that $\mathbb{K}$ is actually easy to use, in spite of the intricate $\mathbb{K}$ concurrent abstract machine technicalities discussed in Section 5.4 — indeed, users of the $\mathbb{K}$ framework need not be familiar with all those intricate details, the same way users of a concurrent programming language need not be aware of the underlying details of the concurrent computing architecture on which their programs are executed. In fact, in this section we make no distinction between the $\mathbb{K}$ rewrite abstract machine and the $\mathbb{K}$ technique, refering to these collectively as "the $\mathbb{K}$ framework", or more simply just "$\mathbb{K}$".

Programming languages, calculi, as well as type systems or formal analyzers can be defined in $\mathbb{K}$ by making use of special, potentially nested *(K) cell* structures, and *(K) (rewrite) rules*. There are two types of $\mathbb{K}$ rules: *computational rules*, which count as computational steps, and *structural rules*,

which do not count as computational steps. The role of the structural rules is to rearrange the term so that the computational rules can apply. $\mathbb{K}$ rules are *unconditional* (they can be thought of as rule schemata and may have ordinary side conditions, though), and they are *context-insensitive*, so $\mathbb{K}$ rules apply concurrently as soon as they match, without any contextual delay or restrictions.

One sort has a special meaning in $\mathbb{K}$, namely the sort K of *computations*, which is also what suggested the name of the framework. The intuition for terms of sort K is that they have computational contents, such as programs or fragments of programs have; indeed, computations extend the syntax of the original language. Computations have a list structure with "$\curvearrowright$" (read "followed by") concatenating two computations and "$\cdot$" the empty computation; the list structure captures the intuition of computation sequentialization. Computations give an elegant and uniform means to define and handle evaluation contexts (Section 3.8.1) and/or continuations [74]. Indeed, a computation "$v \curvearrowright c$" can be thought of as "$c[v]$, that is, evaluation context c applied to v" or as "passing v to continuation c". Computations can be handled like any other terms in a rewriting environment, that is, they can be matched, moved from one place to another in the original term, modified, or even deleted. A term may contain an arbitrary number of computations, which can evolve concurrently; they can be thought of as execution threads. Rules corresponding to inherently sequential operations (such as lookup/assignment of variables in the same thread) must be designed with care, to ensure that they are applied only at the top of computations.

The distinctive feature of $\mathbb{K}$ compared to other term rewriting approaches in general and to rewriting logic (Section 2.7) in particular, is that $\mathbb{K}$ allows rewrite rules to apply *concurrently* even in cases when they overlap, provided that they do not change the overlapped portion of the term. This allows for *truly concurrent semantics* to programming languages and calculi. For example, two threads that read the same location of memory can do that concurrently, even though the corresponding rules overlap on the store location being read. The distinctive feature of $\mathbb{K}$ compared to other frameworks for true concurrency, like chemical abstract machines (Section 3.6.1) or membrane systems (Section 9.7), is that rewrite rules can match across and inside multiple cells and thus perform changes many places at the same time, in one concurrent step.

$\mathbb{K}$ achieves, in one uniform framework, the benefits of both the chemical abstract machines (CHAMs; Section 3.6.1) and reduction semantics with evaluation contexts (RSEC; Section 3.8.1), at the same time avoiding what might be called the "rigidity to chemistry" of the former and the "rigidity to syntax" of the latter. Any CHAM and any RSEC definition can be captured in $\mathbb{K}$ with minimal (in our view *zero*) representational distance. $\mathbb{K}$ can support concurrent language definitions with either an interleaving or a true concurrency semantics.

Like the other semantic approaches that can be represented in rewriting logic (Chapter 3), $\mathbb{K}$ can also be represented in rewriting logic and thus $\mathbb{K}$ definitions can be executed on existing rewrite engines, thus providing "interpreters for free" directly from formal language definitions; additionally, general-purpose formal analysis techniques and tools developed for rewriting logic, such as state space exploration for safety violations or model-checking, give us corresponding techniques and tools for the defined languages, at no additional development cost. Unlike the other semantic approaches (except for the CHAM) whose representations in rewriting logic are *faithful*, in that the resulting rewriting logic theories are step-for-step equivalent with the original definitions, $\mathbb{K}$ cannot be captured faithfully by rewriting logic in any natural way. On the one hand, there is no clear way to represent the $\mathbb{K}$ structural rules in rewriting logic in a general way that properly captures the intuition that the $\mathbb{K}$ computational rules take place "modulo" the structural ones; this results in the rewriting logic theory to potentially miss some of the non-deterministic behaviors of the original $\mathbb{K}$

Original language syntax	K Strictness	K Semantics
$AExp ::= Int$		
$\mid Id$		$\langle \underline{x} \cdots \rangle_{\mathsf{k}}\ \langle \cdots x \mapsto i \cdots \rangle_{\mathsf{state}}$ $\underline{i}$
$\mid AExp$ `+` $AExp$	$[strict]$	i_1 `+` $i_2 \rightarrow i_1 +_{Int} i_2$
$\mid AExp$ `/` $AExp$	$[strict]$	i_1 `/` $i_2 \rightarrow i_1 /_{Int} i_2$ where $i_2 \neq 0$
$BExp ::= Bool$		
$\mid AExp$ `<=` $AExp$	$[seqstrict]$	i_1 `<=` $i_2 \rightarrow i_1 \leq_{Int} i_2$
$\mid$ `not` $BExp$	$[strict]$	`not` $t \rightarrow \neg_{Bool} t$
$\mid BExp$ `and` $BExp$	$[strict(1)]$	$true$ `and` $b \rightarrow b$
		$false$ `and` $b \rightarrow$ `false`
$Stmt ::=$ `skip`		`skip` $\rightarrow \cdot$
$\mid Id$ `:=` $AExp$	$[strict(2)]$	$\langle \underline{x := i} \cdots \rangle_{\mathsf{k}}\ \langle \cdots x \mapsto \underline{\ } \cdots \rangle_{\mathsf{state}}$ $\cdot$ $\underline{i}$
$\mid Stmt$ `;` $Stmt$		s_1 `;` $s_2 \rightharpoonup s_1 \curvearrowright s_2$
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$	$[strict(1)]$	`if` $true$ `then` s_1 `else` $s_2 \rightarrow s_1$
		`if` $false$ `then` s_1 `else` $s_2 \rightarrow s_2$
$\mid$ `while` $BExp$ `do` $Stmt$		$\langle$ `while` b `do` s $\cdots\rangle_{\mathsf{k}}$ `if` b `then` $(s$ `;` `while` b `do` $s)$ `else` $\cdot$
$Pgm ::=$ `vars` **List**`{`Id`}` `;` $Stmt$		$\langle$ `vars` xl `;` $\underline{s}\rangle_{\mathsf{k}}\ \langle \underline{\cdot} \rangle_{\mathsf{state}}$ s $xl \mapsto 0$

Figure 5.1: K definition of IMP: syntax (left), annotations (middle) and semantics (right); $x \in Id$, $xl \in$ **List**`{`Id`}`, $i, i_1, i_2 \in Int$, $t \in Bool$, $b \in BExp$, $s, s_1, s_2 \in Stmt$ (b, s, s_1, s_2 can also be in K)

definition. On the other hand, the corresponding rewriting logic theory may need more interleaved steps in order to capture one concurrent step in the original $\mathbb{K}$ definition; this results in the rewriting logic theory to potentially miss some of the concurrent behaviors of the original $\mathbb{K}$ definition.

5.2.1 $\mathbb{K}$ Semantics of IMP

Figure 5.1 shows the complete $\mathbb{K}$ definition of IMP, except for the configuration; the IMP configuration is explained separately below. The left column in Figure 5.1 gives the IMP syntax (identical to the one in Section 3.1.1), which uses the algebraic CFG notation introduced in Section 2.5. The middle column contains special syntax $\mathbb{K}$ annotations, called *strictness attributes*, stating the evaluation strategy of some language constructs. Finally, the right column gives the semantic rules.

$\mathbb{K}$ makes intensive use of the algebraic CFG notation (Section 2.5) to define configurations, in particular of list, set, multiset and map structures. Like in the CHAM or P-systems, program or system configurations in $\mathbb{K}$ are organized as potentially nested structures of *cells* (we call them cells instead of "molecules" like in CHAM or "membranes" like in P-systems to avoid confusion with terminology in CHAM/P-systems as well as confusion with terminology in chemistry or biology — note that "cell" has a more wide-spread use in computer science, e.g., "memory cell", etc.). However, unlike in CHAM/P-systems which only provide multisets (or bags), $\mathbb{K}$ also provides list, set and map cells in addition to multiset (called bag) cells; $\mathbb{K}$'s cells may be labelled to distinguish them from each other. We use angle brackets as cell wrappers. The $\mathbb{K}$ configuration of IMP can be

defined as follows:

$$\mathit{Configuration}_{\text{IMP}} \quad \equiv \quad \langle \langle K \rangle_{\mathsf{k}} \ \langle \mathbf{Map}\{\mathit{Id} \mapsto \mathit{Int}\} \rangle_{\mathsf{state}} \rangle_{\top}$$

In words, IMP configurations consist of a top cell $\langle \ldots \rangle_{\top}$ containing two other cells inside: a cell $\langle \ldots \rangle_{\mathsf{k}}$ which holds a term of sort K (terms of sort K are called computations and extend the original language syntax as explained in the next paragraph) and and a cell $\langle \ldots \rangle_{\mathsf{state}}$ which holds a map from variables to integers. In Chapter 3, we used the sort *State* as an alias (introduced in Section 3.1.2) for the map sort from variables to integers; since in this chapter we are going to have many maps, lists and bags, we prefer to avoid using aliases. As examples of IMP $\mathbb{K}$ configurations, $\langle \langle \texttt{x := 1; y := x+1} \rangle_{\mathsf{k}} \ \langle \cdot \rangle_{\mathsf{state}} \rangle_{\top}$ is a configuration holding program "`x := 1; y := x+1`" and empty state, and $\langle \langle \texttt{x := 1; y := x+1} \rangle_{\mathsf{k}} \ \langle \texttt{x} \mapsto 0 \ \ \texttt{y} \mapsto 1 \rangle_{\mathsf{state}} \rangle_{\top}$ is a configuration holding the same program and a state mapping `x` to 0 and `y` to 1. When we add threads (in IMP++), the configurations can hold multiple $\langle \ldots \rangle_{\mathsf{k}}$ cells (its bag structure allows that).

$\mathbb{K}$ provides special notational support for *computational structures*, or simply *computations*. Computations have the sort K, which is therefore builtin in the $\mathbb{K}$ framework; the intuition for terms of sort K is that they have computational contents, such as, for example, a program or a fragment of program has. Computations extend the original language/calculus/system syntax with special "$\curvearrowright$"-separated lists "$T_1 \curvearrowright T_2 \curvearrowright \cdots \curvearrowright T_n$" comprising *(computational) tasks*, thought of as having to be "processed" sequentially ("$\curvearrowright$" reads "followed by"). The identity of the "$\curvearrowright$" associative operator is "$\cdot$". Like in reduction semantics with evaluation contexts (RSEC, see Section 3.5), $\mathbb{K}$ allows one to define evaluation contexts over the language syntax. However, unlike in RSEC, parsing does not play any crucial role in $\mathbb{K}$, because $\mathbb{K}$ replaces the hard-to-implement split/plug operations of RSEC by plain, context-insensitive rewriting. Therefore, instead of defining evaluation contexts using grammars and relying on splitting syntactic terms (via parsing) into evaluation contexts and redexes, in $\mathbb{K}$ we define evaluation contexts using special rewrite rules. For example, the evaluation contexts of sum, comparison and conditional in IMP can be defined as follows, by means of *structural rules* (recall that the sum "`+`" was non-deterministic and the comparison "`<=`" was sequential):

$$\begin{aligned}
a_1 \texttt{ + } a_2 &\rightleftharpoons a_1 \curvearrowright \square \texttt{ + } a_2 \\
a_1 \texttt{ + } a_2 &\rightleftharpoons a_2 \curvearrowright a_1 \texttt{ + } \square \\
a_1 \texttt{ <= } a_2 &\rightleftharpoons a_1 \curvearrowright \square \texttt{ <= } a_2 \\
i_1 \texttt{ <= } a_2 &\rightleftharpoons a_2 \curvearrowright i_1 \texttt{ <= } \square \\
\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 &\rightleftharpoons b \curvearrowright \texttt{if } \square \texttt{ then } s_1 \texttt{ else } s_2
\end{aligned}$$

The symbol $\rightleftharpoons$ stands for two structural rules, one left-to-right and another right-to-left.

The right-hand sides of the structural rules above contain, besides the task sequentialization operator "$\curvearrowright$", *freezer* operators containing "$\square$" in their names, such as "$\square$ + _", "_ + $\square$", etc. The first rule above says that in any expression of the form $a_1 \texttt{ + } a_2$, a_1 can be scheduled for processing while a_2 is being held for future processing. Since the rules above are bi-directional, they can be used at will to structurally re-arrange the computations for processing. Thus, when iteratively applied left-to-right they fulfill the role of *splitting* syntax into an evaluation context (the tail of the resulting sequence of computational tasks) and a redex (the head of the resulting sequence), and when applied right-to-left they fulfill the role of *plugging* syntax into context. Such structural rules are often called *heating/cooling rules* in $\mathbb{K}$, because they are reminiscent of the CHAM heating/cooling rules; for example, $a_1 \texttt{ + } a_2$ is "heated" into $a_1 \curvearrowright \square \texttt{ + } a_2$, while $a_1 \curvearrowright \square \texttt{ + } a_2$ is "cooled" into $a_1 \texttt{ + } a_2$. Heating/cooling rules can be used to define any evaluation context, not only strictness of operations.

A language definition can use structural rules not only for heating/cooling but also to give the semantics of some language constructs; this will be discussed later in this section.

To avoid writing obvious heating/cooling structural rules like the above, we prefer to use the *strictness attribute* syntax annotations in $\mathbb{K}$, as shown in the middle column in Figures 5.1 and 5.2: "*strict*" means non-deterministically strict in all enlisted arguments (given by their positions) or by default in all arguments if none enlisted, and "*seqstrict*" is like *strict* but each argument is fully processed before moving to the next one (see the second structural rule of "`<=`" above).

The structural rules corresponding to strictness attributes (or the heating/cooling rules) decompose and eventually push the tasks that are ready for processing to the top (or the left) of the computation. Semantic rules then tell how to process the atomic tasks. The right column in Figure 5.1 shows the semantic $\mathbb{K}$ rules of IMP. To understand them, let us first discuss the important notion of a $\mathbb{K}$ *rule*, which is a strict generalization of the usual notion of a rewrite rule. To take full advantage of $\mathbb{K}$'s support for concurrency, $\mathbb{K}$ rules explicitly mention the parts of the term that they read, write, or don't care about. The underlined parts are those which are written by the rule; the term underneath the line is the new subterm replacing the one above the line.

All writes in a $\mathbb{K}$ rule are applied in *one parallel step*, and, with some reasonable restrictions discussed in Section 5.4 (that avoid read/write and write/write conflicts), writes in multiple $\mathbb{K}$ rule instances can also apply in parallel. The elipses "`...`" represent the volatile part of the term, that is, that part that the current rule does not care about and, consequently, can be concurrently modified by other rules. The operations which are not underlined represent the read-only part of the term: they need to stay unchanged during the application of the rule. For example, the lookup rule in Figure 5.1 (first one) says that once program variable x reaches the top of the computation, it is replaced by the value to which it is mapped in the state, regardless of the remaining computation or the other mappings in the state. Similarly, the assignment rule says that once the assignment statement "x `:=` i" reaches the top of the computation, the value of x in the store is replaced by i and the statement dissolves; in $\mathbb{K}$, "`_`" is a nameless variable of any sort and "$\cdot$" is the unit (or empty) computation (in practice, "$\cdot$" tends to be a polymorphic unit of most if not all list, set and multiset structures). The rule for variable declarations in Figure 5.1 (last one) expects an empty state and allocates and initializes with 0 all the declared variables; the dotted or dashed lines signifies that the rule is structural, which is discussed next.

$\mathbb{K}$ rules are split in two categories: *computational rules* and *structural rules*. Computational rules capture the intuition of computational steps in the execution of the defined system or language, while structural rules capture the intuition of structural rearrangement, rather than computational evolution, of the system. We use dashed or dotted lines in the structural rules to convey the idea that they are lighter-weight than the computational rules. Ordinary rewrite rules are a special case of $\mathbb{K}$ rules, when the entire term is replaced; in this case, we prefer to use the standard notation "$l \to r$" as syntactic sugar for computational rules and the notation "$l \rightharpoondown r$" or "$l \rightharpoonup r$" as syntactic sugar for structural rules. We have seen several structural rules at the beginning of this section, namely the heating/cooling rules corresponding to the strictness attributes. Figure 5.1 shows three more: "s_1 `;` s_2" is rearranged as "$s_1 \curvearrowright s_2$", loops are unrolled when they reach the top of the computation (unconstrained unrolling would lead to undesirable non-termination), and declared variables are allocated in the state. There are no rigid requirements when rules should be computational versus structural and, in the latter case, when one should use "$l \rightharpoondown r$" or "$l \rightharpoonup r$" as syntactic sugar. We (subjectively) prefer to use structural rules for desugaring (like for sequential composition), loop unrolling and declarations, and we prefer to use "$\rightharpoonup$" when syntax is split into computational tasks

Original language syntax	K Strictness	K Semantics
$AExp$ ::= ... | `++` Id		$\langle \underline{\texttt{++}\,x} \curvearrowright \rangle_{\mathsf{k}}\ \langle \curvearrowright x \mapsto \underline{i} \curvearrowright \rangle_{\mathsf{state}}$ $i +_{Int} 1 \qquad\qquad i +_{Int} 1$
$Stmt$ ::= ...		
| `print` $AExp$	[*strict*]	$\langle \underline{\texttt{print}\,i} \curvearrowright \rangle_{\mathsf{k}}\ \langle \curvearrowright \underline{\cdot} \rangle_{\mathsf{output}}$ $\cdot \qquad\qquad i$
| `halt`		$\langle \underline{\texttt{halt} \curvearrowright \_} \rangle_{\mathsf{k}}$ $\cdot$
| `spawn` $Stmt$		$\langle \underline{\texttt{spawn}\,s} \curvearrowright \rangle_{\mathsf{k}}\ \underline{\qquad \cdot \qquad}$ $\cdot \qquad\qquad \langle s \curvearrowright die \rangle_{\mathsf{k}}$
K ::= ... | die		$\langle die \rangle_{\mathsf{k}} \rightharpoonup \cdot$

Figure 5.2: K definition of IMP++ (extends that of IMP in Figure 5.1, *without changing anything*)

and "$\rightharpoonup$" when computational tasks are put back into the original syntax.

Each $\mathbb{K}$ rule can be "desugared" into a standard term rewrite rule by combining all its changes into one top-level change. The relationship between $\mathbb{K}$ rules and conventional term rewriting and rewriting logic is discussed in Section 5.4. The main point is that the resulting rewrite system/theory associated to a $\mathbb{K}$ system lacks some of the potential for concurrency of the original $\mathbb{K}$ system.

5.2.2 $\mathbb{K}$ Semantics of IMP++

In spite of its simplicity, IMP++ revealed limitations in each of the conventional semantic approaches (see Section 3.9); for example, big-step and small-step SOS as well as denotational semantics were heavily non-modular, modular SOS required artificial syntactic extensions of the language in order to attain modularity, reduction semantics with evaluation contexts lacked modularity in some cases, the CHAM relied on a heavy airlock operation to match information in solution molecules, and all these approaches have limited or no support for true concurrency (the CHAM provides more support for true concurrency than the others, but it still unnecessarily enforces interleaving in some cases). In this section we show that $\mathbb{K}$ avoids these limitations, at least in the case of IMP++.

Figure 5.2 shows how the $\mathbb{K}$ semantics of IMP can be seamlessly extended into a semantics for IMP++. To accommodate the output, a new cell needs to be added to the configuration:

$$Configuration_{\text{IMP++}} \quad \equiv \quad \langle \langle K \rangle_{\mathsf{k}}\ \langle \mathbf{Map}\{Id \mapsto Int\} \rangle_{\mathsf{state}}\ \langle \mathbf{List}\{Int\} \rangle_{\mathsf{output}}\ \rangle_{\top}$$

However, note that none of the existing IMP rules needs to change, because each of them only matches what it needs from the configuration. The construct `print` is strict and its rule adds the value of its argument to the end of the output buffer (matches and replaces the unit "$\cdot$" at the end of the buffer). The rule for `halt` disolves the entire computation, and the rule for `spawn` creates a new $\langle \ldots \rangle_{\mathsf{k}}$ cell wrapping the spawned statement. The code in this new cell will be processsed concurrently with the other threads. The last rule "cools" down a terminated thread by simply disolving it; it is a structural rule since, again, we do not want it to count as a computational step.

We conclude this section with a discusion on the concurrency of the $\mathbb{K}$ definition of IMP++. Since in $\mathbb{K}$ rule instances can share read-only data, various (actually all matching) instances of the lookup rule can apply concurrently, in spite of the fact that they overlap on the state subterm. Similarly, since the rules for variable assignment and increment declare volatile everything else in

Original language syntax	K Strictness	K Semantics
$AExp ::= Int$		$i \to int$
$\mid Id$		$\langle \underline{x} \cdots \rangle_{\mathsf{k}}\ \langle \cdots x \cdots \rangle_{\mathsf{vars}}$ int
$\mid AExp$ `+` $AExp$	$[strict]$	int `+` $int \to int$
$\mid AExp$ `/` $AExp$	$[strict]$	int `/` $int \to int$
$\mid$ `++` Id		$\langle \underline{\texttt{++}\, x} \cdots \rangle_{\mathsf{k}}\ \langle \cdots x \cdots \rangle_{\mathsf{vars}}$ int
$BExp ::= AExp$ `<=` $AExp$	$[strict]$	int `<=` $int \to bool$
$\mid$ `not` $BExp$	$[strict]$	`not` $bool \to bool$
$\mid BExp$ `and` $BExp$	$[strict]$	$bool$ `and` $bool \to bool$
$Stmt ::=$ `skip`		`skip` $\to stmt$
$\mid Id$ `:=` $AExp$	$[strict(2)]$	$\langle \underline{x := int} \cdots \rangle_{\mathsf{k}}\ \langle \cdots x \cdots \rangle_{\mathsf{vars}}$ $stmt$
$\mid Stmt$ `;` $Stmt$	$[strict]$	$stmt$ `;` $stmt \to stmt$
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$	$[strict]$	`if` $bool$ `then` $stmt$ `else` $stmt \to stmt$
$\mid$ `while` $BExp$ `do` $Stmt$	$[strict]$	`while` $bool$ `do` $stmt \to stmt$
$\mid$ `print` $AExp$	$[strict]$	`print` $int \to stmt$
$\mid$ `halt`		`halt` $\to stmt$
$\mid$ `spawn` $Stmt$	$[strict]$	`spawn` $stmt \to stmt$
$Pgm ::=$ `vars` **List**`{`Id`}` `;` $Stmt$		$\langle \underline{\texttt{vars}\ xl\ ;\ s} \rangle_{\mathsf{k}}\ \langle \underline{\cdot} \rangle_{\mathsf{vars}}$ $s \curvearrowright pgm$ $\quad$ xl $stmt \curvearrowright pgm \to pgm$

Figure 5.3: K type system for IMP++ (and IMP)

the state except the mapping corresponding to the variable, multiple assignments, increments and reads of distinct variables can happen concurrently. However, if two threads want to write the same variable, or if one wants to write it while another wants to read it, then the two corresponding rules need to interleave, because the two rule instances are in a concurrency conflict. Note also that the rule for `print` matches and changes the end of the output cell; that means, in particular, that multiple outputs by various threads need to be interleaved for the same reason as above. On the other hand, the rule for `spawn` matches any empty top-level position and replaces it by the new thread, so threads can spawn threads concurrently. Similarly, multiple threads can be dissolved concurrently when they are done (last "cooling" structural rule). These concurrency aspects of IMP++ are possible to define formally thanks to the specific nature of the $\mathbb{K}$ rules. If instead we used standard rewrite rules instead of $\mathbb{K}$ rules, than many of the concurrent steps above would need to be interleaved because rewrite rule instances which overlap cannot be applied concurrently.

5.2.3 $\mathbb{K}$ Type System for IMP/IMP++

The $\mathbb{K}$ semantics of IMP/IMP++ discussed above can be used to execute even ill-typed IMP/IMP++ programs, which may be considered undesirable by some language designers. Indeed, one may want to define a type checker for a desired typing policy, and then use it to discard as inappropriate programs that do not obey the desired typing policy. In this section we show how to define a type system for IMP/IMP++ using the very same $\mathbb{K}$ framework. The type system is defined like an (executable) semantics of the language, but one in the more abstract domain of types rather than in

the concrete domain of integer and Boolean values. The technique is general and has been used to define more complex type systems, such as higher-order polymorphic ones (see Section 7.16).

The typing policy that we want to enforce on IMP/IMP++ programs is easy: all variables in a program have by default integer type and must be declared, arithmetic/Boolean operations are applied only on expressions of corresponding types, etc. Since programs and fragments of programs are now going to be rewritten into their types, we need to add to computations some basic types. Also, in addition to the computation to be typed, configurations must also hold the declared variables. Thus, we define the following (the "..." in the definition of K includes all the default syntax of computations, such as the original language syntax, "$\curvearrowright$", freezers, etc.):

$$\begin{array}{rcl} K & ::= & \ldots \mid \mathit{int} \mid \mathit{bool} \mid \mathit{stmt} \mid \mathit{pgm} \\ \mathit{Configuration}^{\mathit{Type}}_{\text{IMP++}} & \equiv & \langle \langle K \rangle_{\mathsf{k}} \ \langle \mathbf{List}\{\mathit{Id}\} \rangle_{\mathsf{vars}} \rangle_{\top} \end{array}$$

Figure 5.3 shows the IMP/IMP++ type system as a $\mathbb{K}$ system over such configurations. Constants reduce to their types, and types are straightforwardly propagated through each language construct. Note that almost each language construct is strict now, because we want to type all its arguments in almost all cases in order to apply the typing policy of the construct. Two constructs make exception, namely the increment and the assignment. The typing policy of these constructs is that they take precisely a variable and not something that types to an integer. If we defined, e.g., the assignment strict and with rule "int `:=` $\mathit{int} \rightarrow \mathit{stmt}$", then our type system would allow ill-formed programs like "`x+y := 0`". Note how we defined the typing policy of programs "`vars` xl `;` s": the declared variables xl are stored into the $\langle \ldots \rangle_{\mathsf{vars}}$ cell (which is expected to initially be empty) and the statement is scheduled for typing (using a structural rule), placing a "reminder" in the computation that the pgm type is expected; once/if the statement is correctly typed, the type pgm is generated.

5.3 $\mathbb{K}$ in Rewrite Logic

In this section we discuss easy and intuitive ways to encode $\mathbb{K}$ systems as rewrite systems. Even though we have not defined the $\mathbb{K}$ rewrite abstract machine (Section 5.4) or the $\mathbb{K}$ technique (Section 5.5) in depth yet, the informal presentation of the $\mathbb{K}$ framework in Section 5.2 suffices to give our reader a reasonably good idea of what $\mathbb{K}$ is and how it works. The role of this section is to extend that informal understanding of $\mathbb{K}$ and give the reader a reasonably good idea of how she can write $\mathbb{K}$ definitions in rewriting logic and then use rewrite engines to execute and formally analyze such definitions. To a large extent, this section is equivalent in style and purpose to similar sections in Chapter 3 explaining how each of the conventional semantic approaches can be represented in rewriting logic and then executed using rewrite engines (e.g., Section 3.2.3, 3.3.3, 3.4.2, etc.).

Since $\mathbb{K}$ is a rewrite-based framework, it is conceptually closer to rewriting logic than any of the other semantic frameworks in Chapter 3. In spite of this closeness, however, there appears to be no immediate faithful embedding of K into rewriting logic. This is mainly because of two reasons:

1. Rewrite logic does not provide direct semantic concurrency support for subterm sharing, which has the effect that rewriting logic rules associated to $\mathbb{K}$ rules need to interleave even in situations where their original $\mathbb{K}$ rules could proceed truly concurrently in the $\mathbb{K}$ framework. As an example, suppose that a, a' and b are terms of sort S, that "a, a, b" is a term of sort $\mathbf{Bag\{}S\mathbf{\}}$, and that we have the following $\mathbb{K}$ rule

$$\frac{a\,,b}{a'}$$

which rewrites a to a' in the presence of a (possibly shared) b. Then in $\mathbb{K}$ we can rewrite the term "a, a, b" in one concurrent step to "a', a', b" (see Section 5.4). This example pushes to its essence the practical situation where two threads (here simulated by each of the two a's) proceed concurrently when they only read (but do not change) the shared memory (here simulated by b). We are going to translate such a $\mathbb{K}$ rule into a rewrite rule

$$a, b \to a', b$$

which applies modulo the associativity and commutativity of the binary comma operation that implicitly constructs $\mathbf{Bag\{}S\mathbf{\}}$. This rewrite rule lacks the concurrency of the original $\mathbb{K}$ rule: no rule instances can overlap in rewriting logic, so two rule instances of the rule above matching the same b cannot proceed concurrently, they need to interleave. All our embeddings of $\mathbb{K}$ into rewriting logic share this limitation, the effect of which is that truly concurrent operations in the original $\mathbb{K}$ semantics become interleaved operations in the resulting rewriting logic theories. Therefore, we cannot obtain a practical concurrent-step-for-concurrent-step faithful embedding of K into rewriting logic[1]. A similar situation was encountered in Section 3.6.1, where the concurrency of rewriting logic could not be simply borrowed to endow, by means of the embedding in Theorem 10, the chemical abstract machine with its desired (claimed but never defined) concurrent semantics.

[1]Theoretically, one could eliminate subterm sharing by subterm copying via equations and multiset rewriting, as discussed in Section 5.4 and in [44], but that is impractical: hard to read/understand the resulting rewrite logic theories encoding the original $\mathbb{K}$ theories, and hard or impossible to execute and to analyze them formally.

2. Rewrite logic does not provide support for structural rules (i.e., rules whose application does not count as computational). In $\mathbb{K}$, computational rules apply "modulo" the structural ones. In other words, given a term t to rewrite using a $\mathbb{K}$ system, the structural rules can be used to derive a set of terms from t, each of those terms being thought of as a "computational representation" of t; then a computational rule can non-deterministically "pick" any term from the set and irreversibly rewrite it to another term. Then the process continues, i.e., the structural rules generate a new set of terms, and so on. Even though this process is reminiscent of how rewrite rules apply in rewriting logic modulo equations, in $\mathbb{K}$ the equations are replaced by structural rules. Metaphorically, one can regard $\mathbb{K}$'s structural rules as "half-equations": they are used like the equations to compute classes of terms on which computational rules apply, but they are not necessarily reversible as the equations are. Indeed, one may have good reasons to not want the structural rules for sequential composition, `while` loops, `vars` declarations in the $\mathbb{K}$ semantics of IMP in Figure 5.1 to be reversible. $\mathbb{K}$'s structural rules are more general than rewriting logic's equations, because one can achieve the same effect of an equation $t = t'$ by replacing it with a pair of structural rules $t \rightleftharpoons t'$. It appears impossible to achieve the desired "modulo" meaning of the structural $\mathbb{K}$ rules in rewriting logic.

Because of the reasons above, in this section we present our embeddings of $\mathbb{K}$ into rewriting logic at a rather conceptual level, discussing for each of them its advantages and disadvantages. The true concurrency and computational granularity aspects tend to be ignored by or unavailable in most conventional semantic frameworks. For example, all the approaches in Chapter 3 except for the chemical abstract machine (CHAM) assume an interleaving semantics, which means that the limitation of the true concurrency of K rewriting to that of rewriting logic is basically meaningless for those approaches, because they lack true concurrency by their very nature. The true concurrency limitation above is also meaningless for the CHAM, because, even though it advocates true concurrency, the CHAM rewriting does not allow for sharing (or, in other words, sharing yields interleaving for the involved rule instances). Therefore, most of the language designers using our embeddings of $\mathbb{K}$ into rewriting logic discussed in this section may not be affected much by the limitations above of the resulting rewriting logic theories. To avoid repetitively mentioning that the resulting rewriting logic theories lack the true concurrency of their original $\mathbb{K}$ definitions, in this section we temporarily restrict $\mathbb{K}$'s concurrency to the one of rewriting logic, that is, the explicit sharing specified in $\mathbb{K}$ rules plays no semantic role in this section.

5.3.1 (Almost Faithful) Embeddings of $\mathbb{K}$ into Rewrite Logic

As briefly discussed in Section 5.2 and in detail discussed in Section 5.4, there are two types of $\mathbb{K}$ rules: structural, whose role is to only rearrange the term without modifying its computational contents, and computational, whose role is to capture the irreversible computational steps. Notationally, the structural rules use dotted lines when written in two-dimensional form and $\multimap$ or $\rightharpoonup$ when written in unidimensional form, and the computational rules use full lines when written two-dimensionally and the usual rewrite relation notation $\rightarrow$ when written in unidimensional form. The unidimensional form is syntactic sugar for the particular case when the entire term is underlined (or rewritten), so we only discuss the more general, two-dimensional representations of the $\mathbb{K}$ rules.

All four embeddings discussed in this section, as well as all the other embeddings that we have

experimented with but do not discuss here, translate $\mathbb{K}$ computational rules of the form

$$p[\underset{r_1}{l_1}, \underset{r_2}{l_2}, \ldots, \underset{r_n}{l_n}]$$

(p is called the local context, or pattern of the $\mathbb{K}$ rule) into corresponding rewrite rules of the form

$$p[l_1, l_2, \ldots, l_n] \to p[r_1, r_2, \ldots, r_n]$$

In order for the above to work, we need to make explicit the anonymous variables ("_") and to desugar the cell comprehension notation (" ⋯ "). For example, the assignment $\mathbb{K}$ rule in Figure 5.1

$$\langle \underset{\cdot}{x := i} \cdots \rangle_{\mathsf{k}} \ \langle \cdots x \mapsto \underset{i}{\_} \cdots \rangle_{\mathsf{state}}$$

is translated into a rewrite rule like the one below

$$\langle X := I \curvearrowright \mathit{Rest} \rangle_{\mathsf{k}} \ \langle X \mapsto J \ \& \ \sigma \rangle_{\mathsf{state}} \to \langle \mathit{Rest} \rangle_{\mathsf{k}} \ \langle X \mapsto I \ \& \ \sigma \rangle_{\mathsf{state}}$$

where the variables have the following sorts: X has sort *Id*; I, J have sort *Int*; *Rest* has sort K; and σ has sort *State*. To respect the well-established convention of rewriting logic, we used capital letters for variables; in $\mathbb{K}$ "paper" definitions we prefer to use lower case letters for variables.

All four embeddings translate structural rules which are not reversible (i.e., structural rules which are not heating/cooling rules) precisely the same way, making therefore no distinction between such structural rules and computational rules in the translation. The only thing which is lost in this translation is the computational granularity of the original $\mathbb{K}$ definition (in addition to the true concurrency of the original $\mathbb{K}$ definition, which we decided to temporarily drop in this section).

What distinguishes the various embeddings of $\mathbb{K}$ into rewriting logic is how they represent the reversible structural rules (the heating/cooling rules). None of the embeddings below is perfect. The first two have a more theoretical relevance than practical, in that the resulting rewriting logic theories are not executable but they capture all the behaviors of the original $\mathbb{K}$ definition (albeit with a different computational granularity than that of the original $\mathbb{K}$ system). The third and fourth embeddings yield executable rewriting logic theories, but their executability comes at a loss of non-deterministic behaviors when executed on current rewriting logic engines. The reason for which we call our embeddings of $\mathbb{K}$ into rewriting logic "almost faithful" is twofold: first, they all miss some of the behaviors of the original $\mathbb{K}$ definition because of the reasons discussed in the preamble of Section 5.3; second, even though our third and fourth embeddings yield executable rewriting logic theories, they actually lose even more of the behaviors of the original $\mathbb{K}$ definition when executed.

First Impractical Embedding of $\mathbb{K}$ into Rewrite Logic

As mentioned, all our embeddings of $\mathbb{K}$ into rewriting logic translate both computational and non-reversible (i.e., non-heating/cooling) structural rules into rewrite rules. Our first embedding takes the simplest and most uniform approach to deal with the remaining reversible structural rules, namely to consider no difference between them and the other $\mathbb{K}$ rules, translating them all into rewrite rules as above. This way, each rewrite sequence in the original $\mathbb{K}$ system can be mirrored into a corresponding sequence in the corresponding rewrite theory and vice-versa, except for the structural versus computational aspect. Therefore, one can use reachability analysis on the resulting

rewrite system, e.g., the search capabilities of Maude, to find all the rewrite sequences that the original $\mathbb{K}$ system can yield. This reachability analysis can be very expensive and impractical, but it is theoretically important to understand that it is possible and that, indeed, the resulting rewrite system does not miss any of the behaviors of the original $\mathbb{K}$ system (assuming the temporarily accepted restrictions and limitations discussed in the preamble of Section 5.3).

There is a big practical problem, though, with this embedding, namely its execution capability. Consider for example a pair of structural (heating/cooling) rules

$$a_1 + a_2 \quad \rightleftharpoons \quad a_1 \curvearrowright \square + a_2$$

which, by this embedding, result into the following rewrite rules:

$$\begin{array}{rcl} a_1 + a_2 & \rightarrow & a_1 \curvearrowright \square + a_2 \\ a_1 \curvearrowright \square + a_2 & \rightarrow & a_1 + a_2 \end{array}$$

These two rules are inverse to each other so they most likely yield infinite rewriting when executed on rewrite engines. Therefore, the rewrite theories resulting from this embedding are not executable.

One could argue that the non-termination problem above is inherent in the original $\mathbb{K}$ definition as well, because there is nothing to stop one from applying the two structural (heating/cooling) rules above indefinitely. While this is true in principle, recall that in $\mathbb{K}$ computational rules apply modulo the structural rules; in particular, one would expect that practical implementations of $\mathbb{K}$ recognize such structural rules and handle them in a special manner. As an analogy, many practical implementations of conventional rewriting provide special support for rewriting *modulo* certain equational properties such as associativity or commutativity; indeed, like our heating/cooling rules above, commutativity would yield infinite rewriting if applied blindly as two rules inverse to each other. Also, like associativity and commutativity, the heating/cooling rules can only yield a finite number of computational representations of a given term, so an implementation can, again in principle, enumerate all of them in order to pick one on which a computational rule can apply.

Second Impractical Embedding of $\mathbb{K}$ into Rewrite Logic

As seen above, simply replacing each structural $\mathbb{K}$ rule by a rewriting logic rule yields non-termination whenever the original $\mathbb{K}$ definition includes any heating/cooling pair of structural rules. Since as discussed above each structural rule can be regarded as a "half-equation" in rewriting logic, it means that each pair of heating/cooling rules $l \rightleftharpoons r$ can be regarded as an equation $l = r$ in rewriting logic. Let us here assume an embedding transformation of K into rewriting logic which translates each pair of heating/cooling rules $l \rightleftharpoons r$ into an equation $l = r$ and each remaining structural $\mathbb{K}$ rule into a rewriting logic rewrite rule as if it was a computational $\mathbb{K}$ rule. Since in rewriting logic the rewrite rules apply modulo the equations, in theory none of the behaviors of the original $\mathbb{K}$ definition is lost.

Like the first embedding above, this embedding is also impractical: its resulting rewriting logic theories are not executable. Indeed, consider the equations

$$\begin{array}{l} a_1 + a_2 = a_1 \curvearrowright \square + a_2 \\ a_1 + a_2 = a_2 \curvearrowright a_1 + \square \end{array}$$

corresponding to the heating/cooling rules for IMP addition in Section 5.2.1, and consider an expression $7 + x$. In order to evaluate it one needs to first lookup x, and in order to do so one needs to heat the expression to $x \curvearrowright 7 + \square$ applying the second equation above. However, if the first equation

is picked by a rewrite engine instead of the second, then the expression is heated to $7 \curvearrowright \square + x$ and now the rewrite process is stuck. Reachability analysis, e.g., using Maude's search command, does not work either, because only the rules are expected to generate new state-space, not the equations, so if the first equation is picked, then the second one will never be tried. The above is enough evidence that this embedding yields non-executable rewrite theories, so it is also impractical. It is interesting to note, as a side point, that the equations corresponding to the heating/cooling rules for non-deterministic constructs like the + above make the resulting rewrite theory non-coherent (see Section 2.7), thus also violating a basic theoretical executability requirement in rewriting logic.

First Practical Embedding of $\mathbb{K}$ into Rewrite Logic

One way to avoid the non-termination problem of the first embedding above is to restrict the applications of the rewrite rules corresponding to the heating/cooling $\mathbb{K}$ rules so that they can only apply in complementary situations. For example, it makes sense to apply the rewrite rule "$a_1 + a_2 \rightarrow a_1 \curvearrowright \square + a_2$" (corresponding to a heating rule) whenever a_1 is not a result, so it needs to be pulled out from its addition context to be further processed, and to apply the rewrite rule "$a_1 \curvearrowright \square + a_2 \rightarrow a_1 + a_2$" (corresponding to a cooling structural rule) whenever a_1 is a result, so it needs no further processing, meaning that it can be plugged back into its original addition context.

No matter how we decide to break the reversibility of the heating/cooling rules by splitting the cases in which only the first rule above applies from the cases in which only the second rule above applies, losing behaviors may be unavoidable. For example, suppose that we split the two cases as above, depending upon whether a_1 is a result or not, and suppose that a_1 is a not a result. Then the first rule applies and a_2 gets frozen until a_1 eventually reduces to an integer, when the second rule plugs it back into the addition context. Now, assuming that the original $\mathbb{K}$ definition also included a heating/cooling structural rule "$a_1 + a_2 \rightleftharpoons a_2 \curvearrowright a_1 + \square$" like the IMP language does (see Section 5.2.1), which is also translated into a pair of rewrite rules like the one for a_1, then the resulting rewrite theory loses those interleaved behaviors in which a_1 is reduced one step, then a_2 is reduced one step, then a_1 is again reduced one step, and so on; the remaining behaviors for + are then only those corresponding to non-deterministic choice: one of its arguments is non-deterministically chosen and evaluated completely, and then the other argument is evaluated completely (these behaviors are the same as those supported by the big-step semantics —Section 3.2).

In many practical situations, the loss of behaviors incurred when switching from fully non-deterministic to non- deterministic choice semantics is acceptable. As already mentioned in Section 3.1, one of the main reasons for which arithmetic language constructs like + are allowed to be non-deterministic is because one wants to allow flexibility in how the language is implemented, and not because these constructs are indeed intended to have fully non-deterministic behaviors. In other words, such constructs are in fact deliberately underspecified and one should not rely on their non-deterministic behavior when writing programs. If one is interested in a faithful rewriting logic semantics that captures even those rare and subtle behaviors that are visible under full non-deterministic but not under non-deterministic choice semantics of arithmetic language constructs, then one is referred to the other three rewriting logic embedding of $\mathbb{K}$ in this section. An alternative is to attempt to statically reject programs containing expressions that can yield such behaviors, the same way a type checker statically rejects programs that do not obey the intended typing policy.

Exercise* 202. *Define a special safety policy for* IMP++ *in* $\mathbb{K}$*, following the type system style in Figure 5.3, which rejects as inappropriate programs containing expressions whose value depends*

upon the particular evaluation strategies of `+`, `/` *and* `<=`. *The* $\mathbb{K}$ *definition of this safety policy should be executable, so that it results into a static analysis tool for this property when executed. For simplicity, the typing policy can be local, that is, should not take into account what other threads can do. Can one define a global policy, where other threads are allowed to potentially interfere, that rejects precisely those programs that violate the desired property and no other programs?*

Once we agree to ignore the loss of behaviors discussed above, we can define an embedding of $\mathbb{K}$ into rewriting logic that takes each pair of heating/cooling structural rules of the form

$$a_1 + a_2 \quad \rightleftharpoons \quad a_2 \curvearrowright a_1 + \square$$

into a pair of potentially conditional[2] rewrite rules

$$\begin{array}{lll} a_1 + a_2 & \rightarrow & a_1 \curvearrowright \square + a_2 \quad \text{when } a_1 \text{ is not a result} \\ a_1 \curvearrowright \square + a_2 & \rightarrow & a_1 + a_2 \quad \text{when } a_1 \text{ is a result} \end{array}$$

The embedding above is easy, mechanical and efficient. Indeed, the rewriting logic definition of IMP obtained by applying the transformation above to the $\mathbb{K}$ definition of IMP in Figure 5.1, when executed in Maude (see Section 5.3.2), yields an interpreter which is faster than any of the Maude interpreters of IMP corresponding to the conventional semantic approaches in Chapter 3 (an even faster interpreter is given by our second practical embedding of $\mathbb{K}$ into rewriting logic discussed next). However, our embedding transformation discussed above still has two limitations:

1. It modifies the computational granularity of the original $\mathbb{K}$ definition, because structural rules that do not count as computational steps in the original $\mathbb{K}$ definition now count as computational rewriting logic steps; and

2. It is rather inefficient when used for exhaustive analysis, e.g., for search or model checking, because one ends up having more rewrite rules like above corresponding to heating/cooling structural $\mathbb{K}$ rules than actual semantic rules (like those in the right columns of the $\mathbb{K}$ definitions of IMP and IMP++ in Figures 5.1 and 5.2), which blow the complexity of the exhaustive analysis tool. Recall from Section 2.7 that rewrite rules are assumed to potentially generate new behaviors, so their application generates the state-space analyzed by such tools, while equations are assumed to not generate new behaviors, so tools apply equations to canonize existing states but not to generate new states.

Second Practical Embedding of $\mathbb{K}$ into Rewrite Logic

We next describe our fourth and in our view best embedding of $\mathbb{K}$ into rewriting logic. It is a combination of the second impractical and the first practical embeddings above. More precisely, instead of transforming the heating/cooling structural rules into pairs of conditional rules as the first practical embedding above does, it transforms them into pairs of equations. For example, the heating/cooling pair "$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$" for $+$ above yields the following two equations:

$$\begin{array}{lll} a_1 + a_2 & = & a_1 \curvearrowright \square + a_2 \quad \text{when } a_1 \text{ is not a result} \\ a_1 \curvearrowright \square + a_2 & = & a_1 + a_2 \quad \text{when } a_1 \text{ is a result} \end{array}$$

[2] One may be able to use subsorting of result and non-result computations into K and thus avoid the conditions.

From a (model- or proof-)theoretical point of view, since the two equations "$l = r$ when ..." and "$r = l$ when ..." associated to a heating/cooling pair of rules "$l \rightleftharpoons r$" have complementary conditions, they are completely equivalent to only one equation, namely "$l = r$". Therefore, from the same theoretical point of view, the embedding discussed here yields rewriting logic theories that are equivalent to the ones yielded by the second impractical embedding above, which means, in particular, that none of the behaviors of the original $\mathbb{K}$ definition is lost.

Like for the other three embeddings of $\mathbb{K}$ into rewriting logic discussed above, the problem with this transformation is also more of a practical rather than theoretical nature. Since the equations are expected to be confluent and to terminate when regarded as rewrite rules and since rewrite rules apply modulo equations, they are not considered as possible sources of non-determinism in current rewrite engines or formal analysis tools. That means that the rewrite theories generated by this new embedding, when executed, lose even more behaviors due to non-deterministic evaluation strategies than the previous embedding. Indeed, instead of non- deterministic choice semantics we now have an "arbitrary but fixed order" semantics: an arbitrary evaluation order is chosen, but one cannot explore any other evaluation order. Note that, in theory, the equations of a rewrite theory need not be confluent (nor terminate) when regarded as rewrite rules, but that in practice they are assumed so by the rewriting logic systems, in that their non-determinism is not explored. Current rewriting logic systems apply the equations as rewrite rules anyway when executing them (so from an executability point of view they are "half-equations", same as the $\mathbb{K}$ structural rules are intended to be), but, since they are not expected to generate new behaviors, less bookkeeping is needed for them, so they are more efficiently executable than the rewrite rules. Equations should therefore be preferred whenever possible (i.e., whenever they are sound) if efficiency is a concern.

5.3.2 $\mathbb{K}$ Semantics and Type System of IMP in Rewrite Logic

We next exemplify the second practical embedding above by completely defining both the $\mathbb{K}$ semantics and the type system of IMP (see Sections 5.2.1 and 5.2.3) in rewriting logic. Although the rewriting logic embeddings of $\mathbb{K}$ discussed above are conceptually straightforward, there are, however, several technical details that need to be addressed in order to make them work. We only focus on the last embedding above here, because, as mentioned, that is the most practical one.

First, we need to introduce the sort K for computations as a list sort with constructors "$\curvearrowright$" for concatenation of computations and "$\cdot$" for the empty computation, that is, with the notation for algebraic context-free grammars in Section 2.5, "$K ::= \mathbf{List}^{\cdot}_{\curvearrowright}\{K\}$". Further, as already mentioned in Section 5.2, K is a supersort for all the syntactic categories; in our case that means that we need to define the subsorts "$AExp, BExp, Stmt, Pgm, \mathbf{List}\{Id\} < K$". Also, we need to define all the necessary computation freezers, e.g., "$\square + \_ : K \to K$", "$\_ + \square : K \to K$", etc., so that the heating/cooling equations parse. To state the conditions of the heating/cooling equations, we also need to define our result computations, namely the elements of sort *KResult*, where $KResult < K$. In the case of the $\mathbb{K}$ semantics of IMP, the results are the integer and the Boolean values, so we add the subsortings "$Int, Bool < KResult$". In the case of the $\mathbb{K}$ type system of IMP, the results are the actual types, namely *int*, *bool*, *stmt*, *pgm*, which we define as constants of sort *KResult*.

All these are shown in Figures 5.4 and 5.5. The type system needs more freezers and more heating/cooling equations than the semantics, because the language constructs are strict in more arguments in the type system than in the semantics. Also, note that we took advantage of two rewrite-logic-specific features in the heating/cooling equations, namely: we used membership assertions in the conditions of the "heating" equations, e.g., "$K_1 : KResult$", and we used unconditional "cooling"

sorts:

$K = \mathbf{List}^{\curvearrowright}\{K\}$, *KResult*, *Cell*

subsorts:

$AExp, BExp, Stmt, Pgm, \mathbf{List}\{Id\} < K$ $\qquad$ $Int, Bool < KResult < K$

operations:

$\langle\_\rangle_{\top} : \mathbf{Bag}\{Cell\} \to Cell$ $\qquad$ $\langle\_\rangle_{\mathsf{k}} : K \to Cell$ $\qquad$ $\langle\_\rangle_{\mathsf{state}} : State \to Cell$

$\square + \_ : K \to K$ $\qquad$ $\_ + \square : K \to K$

$\square \,/\, \_ : K \to K$ $\qquad$ $\_ \,/\, \square : K \to K$

$\square \,\texttt{<=}\, \_ : K \to K$ $\qquad$ $\_ \,\texttt{<=}\, \square : K \to K$

$\square \,\texttt{and}\, \_ : K \to K$

$\texttt{not}\, \square : \to K$

$\_ \,\texttt{:=}\, \square : K \to K$

$\texttt{if}\, \square\, \texttt{then}\, \_\, \texttt{else}\, \_ : K \times K \to K$

strictness equations: // mechanically derived

$K_1 + K_2 = K_1 \curvearrowright \square + K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\qquad$ $R_1 \curvearrowright \square + K_2 = R_1 + K_2$

$K_1 + K_2 = K_2 \curvearrowright K_1 + \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad$ $R_2 \curvearrowright K_1 + \square = K_1 + R_2$

$K_1 \,/\, K_2 = K_1 \curvearrowright \square \,/\, K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\qquad$ $R_1 \curvearrowright \square \,/\, K_2 = R_1 \,/\, K_2$

$K_1 \,/\, K_2 = K_2 \curvearrowright K_1 \,/\, \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad$ $R_2 \curvearrowright K_1 \,/\, \square = K_1 \,/\, R_2$

$K_1 \,\texttt{<=}\, K_2 = K_1 \curvearrowright \square \,\texttt{<=}\, K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\qquad$ $R_1 \curvearrowright \square \,\texttt{<=}\, K_2 = R_1 \,\texttt{<=}\, K_2$

$R_1 \,\texttt{<=}\, K_2 = K_2 \curvearrowright R_1 \,\texttt{<=}\, \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad$ $R_2 \curvearrowright R_1 \,\texttt{<=}\, \square = R_1 \,\texttt{<=}\, R_2$

$K_1 \,\texttt{and}\, K_2 = K_1 \curvearrowright \square \,\texttt{and}\, K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\qquad$ $R_1 \curvearrowright \square \,\texttt{and}\, K_2 = R_1 \,\texttt{and}\, K_2$

$\texttt{not}\, K = K \curvearrowright \texttt{not}\, \square$ **if** $\neg_{Bool}(K : KResult)$ $\qquad$ $R \curvearrowright \texttt{not}\, \square = \texttt{not}\, R$

$K_1 \,\texttt{:=}\, K_2 = K_2 \curvearrowright K_1 \,\texttt{:=}\, \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad$ $R_2 \curvearrowright K_1 \,\texttt{:=}\, \square = K_1 \,\texttt{:=}\, R_2$

$\texttt{if}\, K\, \texttt{then}\, K_1\, \texttt{else}\, K_2 = K \curvearrowright \texttt{if}\, \square\, \texttt{then}\, K_1\, \texttt{else}\, K_2$ **if** $\neg_{Bool}(K : KResult)$

$R \curvearrowright \texttt{if}\, \square\, \texttt{then}\, K_1\, \texttt{else}\, K_2 = \texttt{if}\, K\, \texttt{then}\, K_1\, \texttt{else}\, K_2$

semanitc rules:

$\langle X \curvearrowright Rest\rangle_{\mathsf{k}}\ \langle X \mapsto I\ \&\ \sigma\rangle_{\mathsf{state}} \to \langle I \curvearrowright Rest\rangle_{\mathsf{k}}\ \langle X \mapsto I\ \&\ \sigma\rangle_{\mathsf{state}}$

$I_1 + I_2 \to I_1 +_{Int} I_2$

$I_1 \,/\, I_2 \to I_1 /_{Int} I_2$ **if** $I_2 \neq 0$

$I_1 \,\texttt{<=}\, I_2 \to I_1 \leq_{Int} I_2$

$\texttt{true}\ \texttt{and}\, B_2 \to B_2$

$\texttt{false}\ \texttt{and}\, B_2 \to \texttt{false}$

$\texttt{not true} \to \texttt{false}$

$\texttt{not false} \to \texttt{true}$

$\texttt{skip} \to \cdot$

$\langle X \,\texttt{:=}\, I \curvearrowright Rest\rangle_{\mathsf{k}}\ \langle X \mapsto J\ \&\ \sigma\rangle_{\mathsf{state}} \to \langle Rest\rangle_{\mathsf{k}}\ \langle X \mapsto I\ \&\ \sigma\rangle_{\mathsf{state}}$

$S_1\ \texttt{;}\ S_2 \to S_1 \curvearrowright S_2$

$\texttt{if true then}\, S_1\, \texttt{else}\, S_2 \to S_1$

$\texttt{if false then}\, S_1\, \texttt{else}\, S_2 \to S_2$

$\langle\, \texttt{while}\, B\, \texttt{do}\, S \curvearrowright Rest\rangle_{\mathsf{k}} \to \langle\, \texttt{if}\, B\, \texttt{then}\, (S\ \texttt{;}\ \texttt{while}\, B\, \texttt{do}\, S)\, \texttt{else skip} \curvearrowright Rest\rangle_{\mathsf{k}}$

$\langle\, \texttt{vars}\, Xl\ \texttt{;}\ S\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{state}} \to \langle S\rangle_{\mathsf{k}}\ \langle Xl \mapsto 0\rangle_{\mathsf{state}}$

Figure 5.4: Complete $\mathbb{K}$ semantics of IMP in rewriting logic (variables K, K_1, K_2 B, B_2, S, S_1, S_2, *Rest* have *kind* $[K]$, variables R, R_1 and R_2 have sort *KResult*, variable X has sort *Id*, variable Xl has sort $\mathbf{List}\{Id\}$, variable σ has sort *State*, and variables I, I_1, I_2, J have sort *Int*).

sorts:

$K = \mathbf{List}^{\cdot}_{\curvearrowright}\{K\}$, $KResult$, $Cell$

subsorts:

$AExp, BExp, Stmt, Pgm, \mathbf{List}\{Id\} < K$

operations:

$int, bool, stmt, pgm : \to KResult$ // result constants, corresponding to the types

$\langle\_\rangle_{\top} : \mathbf{Bag}\{Cell\} \to Cell$ $\quad \langle\_\rangle_{\mathsf{k}} : K \to Cell$ $\quad \langle\_\rangle_{\mathsf{vars}} : \mathbf{List}\{Id\} \to Cell$

// all operations whose names contain $\square$ are mechanically derived from strictness attributes

$\square + \_ : K \to K$ $\quad \_ + \square : K \to K$

$\square / \_ : K \to K$ $\quad \_ / \square : K \to K$

$\square \texttt{<=} \_ : K \to K$ $\quad \_ \texttt{<=} \square : K \to K$

$\square \texttt{ and } \_ : K \to K$ $\quad \_ \texttt{ and } \square : K \to K$

$\texttt{not } \square : \to K$

$\_ \texttt{:=} \square : K \to K$ $\quad \texttt{if } \square \texttt{ then } \_ \texttt{ else } \_ : K \times K \to K$

$\texttt{if } \_ \texttt{ then } \square \texttt{ else } \_ : K \times K \to K$ $\quad \texttt{if } \_ \texttt{ then } \_ \texttt{ else } \square : K \times K \to K$

$\texttt{while } \square \texttt{ do } \_ : K \to K$ $\quad \texttt{while } \_ \texttt{ do } \square : K \to K$

strictness equations:

// all equations below are mechanically derived from the strictness attributes

$K_1 + K_2 = K_1 \curvearrowright \square + K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\quad R_1 \curvearrowright \square + K_2 = R_1 + K_2$

$K_1 + K_2 = K_2 \curvearrowright K_1 + \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\quad R_2 \curvearrowright K_1 + \square = K_1 + R_2$

$K_1 \,/\, K_2 = K_1 \curvearrowright \square \,/\, K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\quad R_1 \curvearrowright \square \,/\, K_2 = R_1 \,/\, K_2$

$K_1 \,/\, K_2 = K_2 \curvearrowright K_1 \,/\, \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\quad R_2 \curvearrowright K_1 \,/\, \square = K_1 \,/\, R_2$

$K_1 \texttt{ <= } K_2 = K_1 \curvearrowright \square \texttt{ <= } K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\quad R_1 \curvearrowright \square \texttt{ <= } K_2 = R_1 \texttt{ <= } K_2$

$K_1 \texttt{ <= } K_2 = K_2 \curvearrowright K_1 \texttt{ <= } \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\quad R_2 \curvearrowright K_1 \texttt{ <= } \square = K_1 \texttt{ <= } R_2$

$K_1 \texttt{ and } K_2 = K_1 \curvearrowright \square \texttt{ and } K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\quad R_1 \curvearrowright \square \texttt{ and } K_2 = R_1 \texttt{ and } K_2$

$K_1 \texttt{ and } K_2 = K_2 \curvearrowright K_1 \texttt{ and } \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\quad R_2 \curvearrowright K_1 \texttt{ and } \square = K_1 \texttt{ and } R_2$

$\texttt{not } K = K \curvearrowright \texttt{not } \square$ **if** $\neg_{Bool}(K : KResult)$ $\quad R \curvearrowright \texttt{not } \square = \texttt{not } R$

$K_1 \texttt{ := } K_2 = K_2 \curvearrowright K_1 \texttt{ := } \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\quad R_2 \curvearrowright K_1 \texttt{ := } \square = K_1 \texttt{ := } R_2$

$K_1 \texttt{ ; } K_2 = K_1 \curvearrowright \square \texttt{ ; } K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\quad R_1 \curvearrowright \square \texttt{ ; } K_2 = R_1 \texttt{ ; } K_2$

$K_1 \texttt{ ; } K_2 = K_2 \curvearrowright K_1 \texttt{ ; } \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\quad R_2 \curvearrowright K_1 \texttt{ ; } \square = K_1 \texttt{ ; } R_2$

$\texttt{if } K \texttt{ then } K_1 \texttt{ else } K_2 = K \curvearrowright \texttt{if } \square \texttt{ then } K_1 \texttt{ else } K_2$ **if** $\neg_{Bool}(K : KResult)$

$R \curvearrowright \texttt{if } \square \texttt{ then } K_1 \texttt{ else } K_2 = \texttt{if } K \texttt{ then } K_1 \texttt{ else } K_2$

$\texttt{if } K \texttt{ then } K_1 \texttt{ else } K_2 = K_1 \curvearrowright \texttt{if } K \texttt{ then } \square \texttt{ else } K_2$ **if** $\neg_{Bool}(K_1 : KResult)$

$R_1 \curvearrowright \texttt{if } K \texttt{ then } \square \texttt{ else } K_2 = \texttt{if } K \texttt{ then } R_1 \texttt{ else } K_2$

$\texttt{if } K \texttt{ then } K_1 \texttt{ else } K_2 = K_2 \curvearrowright \texttt{if } K \texttt{ then } K_1 \texttt{ else } \square$ **if** $\neg_{Bool}(K_2 : KResult)$

$R_2 \curvearrowright \texttt{if } K \texttt{ then } K_1 \texttt{ else } \square = \texttt{if } K \texttt{ then } K_1 \texttt{ else } R_2$

$\texttt{while } K_1 \texttt{ do } K_2 = K_1 \curvearrowright \texttt{while } \square \texttt{ do } K_2$ **if** $\neg_{Bool}(K_1 : KResult)$

$R_1 \curvearrowright \texttt{while } \square \texttt{ do } K_2 = \texttt{while } R_1 \texttt{ do } K_2$

$\texttt{while } K_1 \texttt{ do } K_2 = K_2 \curvearrowright \texttt{while } K_1 \texttt{ do } \square$ **if** $\neg_{Bool}(K_2 : KResult)$

$R_2 \curvearrowright \texttt{while } K_1 \texttt{ do } \square = \texttt{while } K_1 \texttt{ do } R_2$

Figure 5.5: $\mathbb{K}$ computations, configurations, and strictness attributes for the definition of IMP's type system in rewriting logic; the remaining semantic rules and equations are given in Figure 5.6 (variables K, K_1, K_2 have *kind* $[K]$, and variables R, R_1 and R_2 have sort $KResult$).

semanitc rules:

$\langle X \curvearrowright Rest \rangle_{\mathsf{k}}\ \langle Xl, X, Xl' \rangle_{\mathsf{vars}} \to \langle int \curvearrowright Rest \rangle_{\mathsf{k}}\ \langle Xl, X, Xl' \rangle_{\mathsf{vars}}$

int `+` $int \to int$

int `/` $int \to int$

int `<=` $int \to int$

$bool$ `and` $bool \to bool$

`not` $bool \to bool$

`skip` $\to stmt$

$\langle X$ `:=` $int \curvearrowright Rest \rangle_{\mathsf{k}}\ \langle Xl, X, Xl' \rangle_{\mathsf{vars}} \to \langle stmt \curvearrowright Rest \rangle_{\mathsf{k}}\ \langle Xl, X, Xl' \rangle_{\mathsf{vars}}$

$stmt$ `;` $stmt \to stmt$

`if` $bool$ `then` $stmt$ `else` $stmt \to stmt$

`while` $bool$ `do` $stmt \to stmt$

$\langle$ `vars` Xl `;` $S \rangle_{\mathsf{k}}\ \langle \cdot \rangle_{\mathsf{vars}} \to \langle S \curvearrowright pgm \rangle_{\mathsf{k}}\ \langle Xl \rangle_{\mathsf{vars}}$

$stmt \curvearrowright pgm \to pgm$

Figure 5.6: The semantic rules of the $\mathbb{K}$ definition of IMP's type system in rewriting logic (variable X has sort *Id*, variables Xl and Xl' have sort $\mathbf{List\{}Id\mathbf{\}}$, variable I has sort *Int*, and variables S and *Rest* have *kind* $[K]$)

equations but ones using variables of sort *KResult*. In rewriting logic we can assume that all the sorts of all the terms are dynamically computed and known at any moment (a term can have more than one sort, because of subsorting and operator overloading). The membership assertions allow one to dynamically check whether a term has a desired sort. If one does not want to rely on membership assertions and subsorting, e.g., if one's target engine does not support these, then one can alternatively define one's own membership predicate and make both equations conditional.

The next step is to define the cell-based configurations. Both the $\mathbb{K}$ semantics and the type system of IMP admit very simple configurations, consisting of a top cell that contains two subcells, the computation and either the state (in the semantics) or the list of variables (in the type system). These are defined by means of three operations, listed in the first raw of operations in Figures 5.4 and 5.5. The semantic rules in Figures 5.4 and 5.6 are straightforward: they are obtained by blindly applying the transformation discussed in the preamble of Section 5.3.1 to the corresponding $\mathbb{K}$ rules in Figures 5.1 and 5.3. Note that both the $\mathbb{K}$ computational rules and the non-reversible structural rules were translated into rewrite rules. The distinction between the two categories of $\mathbb{K}$ rules is therefore "lost in translation"; the resulting rewrite theories have finer-grained computational steps.

Note that the non-result $\mathbb{K}$ variables in Figures 5.4, 5.5, and 5.6 actually were assumed to have the *kind* $[K]$ and not the sort K. The reason is that the strictness equations corresponding to the heating structural rules can apply anywhere, including inside arguments of language constructs. When that happens, the respective arguments change their sort from their original language syntactic sort into K. Since the respective language construct expected the original syntactic sort which is subsorted to K and not K, the resulting term will therefore end up having the kind $[K]$. For more on the relationship between sorts, subsorts and kinds, the reader is referred to Section 2.7.

```
mod K-COMPUTATION is
  sorts K KResult .  subsort KResult < K .
  op .K : -> K .
  op _~>_ : K K -> K [assoc id: .K] .
endm

mod K-CONFIGURATION is including K-COMPUTATION .
  sorts Cell Bag{Cell} .  subsort Cell < Bag{Cell} .
  op .Bag{Cell} : -> Bag{Cell} .
  op __ : Bag{Cell} Bag{Cell} -> Bag{Cell} [assoc comm id: .Bag{Cell}] .
  op <T>_</T> : Bag{Cell} -> Cell .
  op <k>_</k> : K -> Cell .
endm
```

Figure 5.7: Generic $\mathbb{K}$ computations and configurations in Maude

☆ $\mathbb{K}$ Semantics and Type System of IMP in Maude

Here we discuss the Maude representations of the rewrite theories above. The only notable difference between the next Maude modules and the rewrite theories above is that the various list and bag sorts, which were simply assumed above, need to be explicitly defined as associative and associative/commutative operations in Maude.

Figure 5.7 shows generic Maude definitions of $\mathbb{K}$ computations and configurations, which can be used across many $\mathbb{K}$ definitions, including both the IMP semantics and the IMP type system discussed here. Concrete definitions subsort their syntax to K and may define additional cells. To distinguish the unit or empty computation "$\cdot$" from other empty or unit constants, we follow our general convention in this book and write it "`.K`" (a dot followed by its sort). We follow the same convention for the empty cell, namely we write it "`.Bag{Cell}`".

Figure 5.8 shows the Maude definition of the $\mathbb{K}$ strictness attributes corresponding to the $\mathbb{K}$ rewriting logic definition of IMP in Figure 5.4. Note that, for the reason explained above, the variables `K`, `K1` and `K2` are declared to have the kind `[K]`. The Maude modules in Figure 5.8 are admittedly low level and boring to define. Indeed, the user of K-Maude (see Section 5.6) will never define them; instead, she only adds strictness attributes to syntactic language constructs, like we did in Figure 5.1. Nevertheless, if one wants to write language definitions using the $\mathbb{K}$ semantic technique in plain Maude, without relying on any other tools (the same way we wrote Maude language definitions using various semantic techniques in Chapter 3), then, unfortunately, one has to manually define such low level operations and equations (similarly, recall that one had to manually define the infrastructure for splitting/plugging in RSEC in Section 3.5). When defining the strictness equations, one will most likely use cut-and-paste; one should be careful to replace all the symbols appropriately (e.g., the `+` into `/`, etc.), otherwise one's language may have hard to debug errors, such as performing an operation instead of another one.

To test the strictness equations, one can ask Maude to rewrite various programs or fragments of program. For example, the rewrite command

```
Maude> rewrite if 3 <= (2 + x) / 7 then x := 3 / x else x := x / 7 ; y := x .
```

yields the following result:

```
rewrites: 45 in 0ms cpu (0ms real) (0 rewrites/second)
result [K]: x ~> 2 +[] ~> []/ 7 ~> 3 <=[]
            ~> if [] then x ~> 3 / [] ~> x := [] else x ~> [] / 7 ~> x := [] ; x ~> y := []
```

```
mod IMP-STRICTNESS-K is including K-COMPUTATION + IMP-SYNTAX .
  subsorts AExp BExp Stmt Pgm List{Id} < K .
  subsorts Int Bool < KResult .

  var K K1 K2 : [K] .  var R R1 R2 : KResult .

  ops ([]+_) (_+[]) : K -> K .
 ceq K1 + K2 = K1 ~> [] + K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] + K2 = R1 + K2 .
 ceq K1 + K2 = K2 ~> K1 + [] if notBool(K2 :: KResult) .
  eq R2 ~> K1 + [] = K1 + R2 .

  ops ([]/_) (_/[]) : K -> K .
 ceq K1 / K2 = K1 ~> [] / K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] / K2 = R1 / K2 .
 ceq K1 / K2 = K2 ~> K1 / [] if notBool(K2 :: KResult) .
  eq R2 ~> K1 / [] = K1 / R2 .

  ops ([]<=_) (_<=[]) : K -> K .
 ceq K1 <= K2 = K1 ~> [] <= K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] <= K2 = R1 <= K2 .
 ceq R1 <= K2 = K2 ~> R1 <= [] if notBool(K2 :: KResult) .
  eq R2 ~> R1 <= [] = R1 <= R2 .

  op []and_ : K -> K .
 ceq K1 and K2 = K1 ~> [] and K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] and K2 = R1 and K2 .

  op not[] : -> K .
 ceq not K = K ~> not [] if notBool(K :: KResult) .
  eq R ~> not [] = not R .

  op _:=[] : K -> K .
 ceq K1 := K2 = K2 ~> K1 :=[] if notBool(K2 :: KResult) .
  eq R2 ~> K1 :=[] = K1 := R2 .

  op if[]then_else_ : K K -> K .
 ceq if K then K1 else K2 = K ~> if[]then K1 else K2 if notBool(K :: KResult) .
  eq R ~> if[]then K1 else K2 = if R then K1 else K2 .
endm
```

Figure 5.8: $\mathbb{K}$ strictness attributes of IMP in Maude

```
mod IMP-SEMANTICS-K is including IMP-STRICTNESS-K + K-CONFIGURATION + STATE .
  op <state>_</state> : State -> Cell .
  var X : Id .  var Xl : List{Id} .  var Sigma : State .
  var I I1 I2 J : Int .  var B B2 S S1 S2 K Rest : [K] .
  rl <k> X ~> Rest </k> <state> X |-> I & Sigma </state>
  => <k> I ~> Rest </k> <state> X |-> I & Sigma </state> .
  rl I1 + I2 => I1 +Int I2 .
 crl I1 / I2 => I1 /Int I2 if I2 =/=Bool 0 .
  rl I1 <= I2 => I1 <=Int I2 .
  rl true and B2 => B2 .
  rl false and B2 => false .
  rl not true => false .
  rl not false => true .
  rl skip => .K .
  rl <k> X := I ~> Rest </k> <state> X |-> J & Sigma </state>
  => <k>           Rest </k> <state> X |-> I & Sigma </state> .
  eq S1 ; S2 = S1 ~> S2 .
  rl if true  then S1 else S2 => S1 .
  rl if false then S1 else S2 => S2 .
  eq <k> (while B do S) ~> Rest </k> = <k> (if B then S ; while B do S else skip) ~> Rest </k> .
  eq <k> vars Xl ; S </k> <state> .State </state> = <k> S </k> <state> Xl |-> 0 </state> .
endm
```

Figure 5.9: $\mathbb{K}$ semantics of IMP in Maude

Note that, unlike in the CHAM where the rules only apply in solutions (see Section 3.6.1), the strictness (heating) equations were applied everywhere they matched. As a result, the heated term does not have the sort *Stmt* anymore, not even the sort K, but the kind $[K]$. The reason is that the sequential composition construct, "`;`", now takes two terms of sort K instead of two terms of sort *Stmt*, so it cannot yield a well-sorted term. This is also the reason for which all the variables ranging over computations were and will continue to be defined to have kind $[K]$ (as opposed to sort K, to accommodate the fact that the strictness equations can eagerly apply anywhere transforming syntactic terms into computations.

Once the low-level, mechanical and tedious $\mathbb{K}$ infrastructure and strictness equations are defined, the interesting Maude rules corresponding to the semantic $\mathbb{K}$ rules are natural and elegant. The module in Figure 5.9 contains all the Maude rewrite rules corresponding to actual $\mathbb{K}$ semantic rules in the definition of IMP. Like for all the semantic approaches in Chapter 3, Maude, through its rewriting capabilities, gives us an IMP interpreter by simply executing the semantics discussed above. For example, the Maude rewrite command

```
Maude> rewrite <T> <k> sumPgm </k> <state> .State </state> </T> .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are also irrelevant, so they were replaced by "..."):

```
rewrites: 6566 in ... cpu (... real) (... rewrites/second)
result Cell: <T> <k> .K </k> <state> n |-> 0 & s |-> 5050 </state> </T>
```

The resulting Maude interpreter is faster than any of the similar interpreters discussed in Chapter 3. We believe that the reason for the increased performance stays in the fact that the resulting rewriting logic rules are mostly unconditional, and unconditional rules generally outperform the conditional

ones (to apply a conditional rule, the rewrite engine needs to stack the current rewrite context, start a new rewrite session to evaluate the condition, then pop the previous context, etc.). The reason for which the reduction semantics with evaluation contexts interpreters obtained in Section 3.5 are much slower, in spite of the fact that the third of them also had mostly unconditional semantic rules, is because their splitting/plugging mechanism had to be defined using conditional rules (actually very expensive ones, which enable full search in their conditions).

One can use any of the general-purpose tools provided by Maude on the $\mathbb{K}$ semantic definition above. For example, one can exhaustively search for all possible behaviors of a program:

```
Maude> search <T> <k> sumPgm </k> <state> .State </state> </T> =>! Cfg:Cell .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic.

Exercise 203. *Modify the Maude code in Figures 5.8 and 5.9 so that / short-circuits when its numerator evaluates to* 0 *(see also Exercises 61, 65, 67, 72, 77, and 82).*

Exercise 204. *Modify the Maude code in Figures 5.8 and 5.9 so that conjunction is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments (see also Exercises 62, 66, 68, 73, 78, and 83).*

Figures 5.10 and 5.11 are the Maude versions of the rewriting logic theories in Figures 5.5 and 5.6 corresponding to the $\mathbb{K}$ type system of IMP. The typing rules in Figure 5.11 are a blind Maude representation of those in Figure 5.6 and are self-explanatory.

```
mod IMP-TYPE-SYSTEM-STRICTNESS-K is including K-COMPUTATION + IMP-SYNTAX .
  subsorts AExp BExp Stmt Pgm List{Id} KResult < K .
  ops int bool stmt pgm : -> KResult .

  var K K1 K2 : [K] .  var R R1 R2 : KResult .

  ops ([]+_) (_+[]) : K -> K .
 ceq K1 + K2 = K1 ~> [] + K2 if notBool(K1 :: KResult) .      eq R1 ~> [] + K2 = R1 + K2 .
 ceq K1 + K2 = K2 ~> K1 + [] if notBool(K2 :: KResult) .      eq R2 ~> K1 + [] = K1 + R2 .

  ops ([]/_) (_/[]) : K -> K .
 ceq K1 / K2 = K1 ~> [] / K2 if notBool(K1 :: KResult) .      eq R1 ~> [] / K2 = R1 / K2 .
 ceq K1 / K2 = K2 ~> K1 / [] if notBool(K2 :: KResult) .      eq R2 ~> K1 / [] = K1 / R2 .

  ops ([]<=_) (_<=[]) : K -> K .
 ceq K1 <= K2 = K1 ~> [] <= K2 if notBool(K1 :: KResult) .    eq R1 ~> [] <= K2 = R1 <= K2 .
 ceq K1 <= K2 = K2 ~> K1 <= [] if notBool(K2 :: KResult) .    eq R2 ~> K1 <= [] = K1 <= R2 .

  ops ([]and_) (_and[]) : K -> K .
 ceq K1 and K2 = K1 ~> [] and K2 if notBool(K1 :: KResult) .  eq R1 ~> [] and K2 = R1 and K2 .
 ceq K1 and K2 = K2 ~> K1 and [] if notBool(K2 :: KResult) .  eq R2 ~> K1 and [] = K1 and R2 .

  op not[] : -> K .
 ceq not K = K ~> not [] if notBool(K :: KResult) .           eq R ~> not [] = not R .

  op _:=[] : K -> K .
 ceq K1 := K2 = K2 ~> K1 :=[] if notBool(K2 :: KResult) .     eq R2 ~> K1 :=[] = K1 := R2 .

  ops ([];_) (_;[]) : K -> K .
 ceq K1 ; K2 = K1 ~> [] ; K2 if notBool(K1 :: KResult) .      eq R1 ~> [] ; K2 = R1 ; K2 .
 ceq K1 ; K2 = K2 ~> K1 ; [] if notBool(K2 :: KResult) .      eq R2 ~> K1 ; [] = K1 ; R2 .

  ops (if[]then_else_) (if_then[]else_) (if_then_else[]) : K K -> K .
 ceq if K then K1 else K2 = K ~> if[]then K1 else K2 if notBool(K :: KResult) .
  eq R ~> if[]then K1 else K2 = if R then K1 else K2 .
 ceq if K then K1 else K2 = K1 ~> if K then [] else K2 if notBool(K1 :: KResult) .
  eq R1 ~> if K then [] else K2 = if K then R1 else K2 .
 ceq if K then K1 else K2 = K2 ~> if K then K1 else [] if notBool(K2 :: KResult) .
  eq R2 ~> if K then K1 else [] = if K then K1 else R2 .

  ops (while[]do_) (while_do[]) : K -> K .
 ceq while K1 do K2 = K1 ~> while [] do K2 if notBool(K1 :: KResult) .
  eq R1 ~> while [] do K2 = while R1 do K2 .
 ceq while K1 do K2 = K2 ~> while K1 do [] if notBool(K2 :: KResult) .
  eq R2 ~> while K1 do [] = while K1 do R2 .
endm
```

Figure 5.10: $\mathbb{K}$ strictness attributes for the definition of the type system of IMP in Maude.

```
mod IMP-TYPE-SYSTEM-SEMANTICS-K is including IMP-TYPE-SYSTEM-STRICTNESS-K + K-CONFIGURATION .
  op <vars>_</vars> : List{Id} -> Cell .
  var X : Id .  var Xl Xl' : List{Id} .  var I : Int .  var S Rest : [K] .
  rl I => int .
  rl <k> X   ~> Rest </k> <vars> Xl, X, Xl' </vars>
  => <k> int ~> Rest </k> <vars> Xl, X, Xl' </vars> .
  rl int + int => int .
  rl int / int => int .
  rl int <= int => bool .
  rl bool and bool => bool .
  rl not bool => bool .
  rl skip => stmt .
  rl <k> X := int ~> Rest </k> <vars> Xl, X, Xl' </vars>
  => <k> stmt     ~> Rest </k> <vars> Xl, X, Xl' </vars> .
  rl stmt ; stmt => stmt .
  rl if bool then stmt else stmt => stmt .
  rl while bool do stmt => stmt .
  eq <k> vars Xl ; S </k> <vars> .List{Id} </vars> = <k> S ~> pgm </k> <vars> Xl </vars> .
  rl stmt ~> pgm => pgm .
endm
```

Figure 5.11: $\mathbb{K}$ type system of IMP in Maude